IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES

Application No.:	10/646,309	Examiner:	Chen, Qing
Atty. Docket No.:	SUN-P9042	Art Unit:	2191
Filed:	22 August 2003	Conf. No.:	9198
Appellant:	Gregory M. Wright et al.		

For: *Reducing The Overhead Involved In Executing Native Code In A Virtual Machine Through Binary Reoptimization*

APPEAL BRIEF

Sir:

In response to the Notice of Panel Decision from Appeal Brief Review mailed 19 April 2010, and subsequent to the Notice of Appeal filed 12 February 2010, Appellant submits this Appeal Brief to appeal the rejection of claims 1-2, 4-11, 13-18, and 28-25 under 35 U.S.C. § 103(a) in a Final Office Action mailed 12 November 2009. This Appeal Brief demonstrates that such rejections cannot be sustained because the Examiner has not established a proper prima facie case of obviousness.

TABLE OF CONTENTS

THE REAL PARTY IN INTEREST

The real party in interest in this appeal is Sun Microsystems, Inc., the assignee of this application.

RELATED APPEALS AND INTERFERENCES

Appellant is not aware of any appeals or interferences that will affect directly, will be affected directly by, or will otherwise have bearing on the decision in this appeal.

STATUS OF CLAIMS

The status of the claims is as follows:

Claims pending:	1-2, 4-11, 13-18, and 28-35
Claims rejected:	1-2, 4-11, 13-18, and 28-35
Claims objected to:	2, 9, 11, and 18
Claims cancelled:	3, 12, 19-27, and 36-39
Claims appealed:	1, 10, 28, and 32

STATUS OF AMENDMENTS

In an amendment filed on 17 May 2010, Appellant amended claims 1-2, 4-6, 9-11, 13-15, 18-, 28-29, 31-33, and 35 to place the claims in better condition for appeal. A copy of the amended claims is attached as Appendix A.

SUMMARY OF THE CLAIMED SUBJECT MATTER

The claims in the instant application are directed toward a method, and a computer-readable storage device for reducing an overhead involved in executing native code methods in an application running on a virtual machine.

As described in the Background section of the instant application, the Java programming language allows applications to be executed by a large number of different computing platforms. Java applications can be compiled into modules containing platform-independent byte codes, which can be executed using a Java virtual machine. In some cases, it is useful for a platform-independent application to access code written in other languages ("native code methods"). Java provides the Java Native Interface (JNI), which enables Java applications to access native code methods. The JNI also provides an interface through which native code methods can manipulate heap objects within the virtual machine. Unfortunately, calls to a native code method can involve a significant amount of overhead. For example, a call to a native code method can involve indirect calls and associated indirect references. These indirect calls and associated references can introduce a significant amount of overhead, which can affect the performance of the Java application.

The claimed embodiments address these problems by selecting a call to any native code method to be optimized within the virtual machine. These embodiments decompile at least part of the native code method for the selected call into an intermediate representation. This intermediate representation includes a set of instruction code which is not in final executable form. Also, the claimed embodiments obtain an intermediate representation for the application running on the virtual machine which interacts with the native code method for the selected call.

The claimed embodiments integrate the intermediate representation for the native code method for the selected call into the intermediate representation for the application to form an integrated intermediate representation. Next, the claimed embodiments generate a native code from the integrated intermediate representation. Generating the native code involves optimizing interactions between the application and the native code method for the selected call. Optimizing the interactions involves optimizing calls from the application to the native code method by using additional information from the integrated intermediate representation. The claimed embodiments use this additional information to reduce a number of indirect calls and indirect references associated with the calls from the application to the native code method for the selected call.

Native code methods are described on page 2, lines 9-14, and page 6, line 19, to page 7, line 4, and are illustrated in FIG. 1 of the instant application. Decompiling into an intermediate representation is described on page 8, lines 1-13 of the instant application. Obtaining an intermediate representation associated with the application is described on page 8, lines 14-20 of the instant application. Integrating the intermediate representation into the intermediate representation is described on page 8, lines 21-26 of the instant application. Generating the native code from the integrated intermediate representation is described on page 9, lines 1-4 of the instant application. Optimizing the interactions is described on page 7, lines 5-7, and page 9, lines 5-12 of the instant application.

<u>Independent Claim 1: A Method For Reducing An Overhead Involved In Executing Native Code Methods In An Application Running On A Virtual Machine</u>

The claimed method ("the method") reduces an overhead involved in executing native code methods in an application running on a virtual machine.

The method is described on page 7, line 10, to page 9, line 4 of the instant application.

The method involves selecting a call to a native code method to be optimized within the virtual machine. Native code methods are described on page 2, lines 9-14, and page 6, line 19, to page 7, line 4, and are illustrated in FIG. 1 of the instant application.

The method further involves decompiling at least part of the native code method for the selected call into an intermediate representation. An intermediate representation includes a set of instruction code which is not in final executable form. Decompiling into an intermediate representation is described on page 8, lines 1-13 of the instant application.

The method further involves obtaining an intermediate representation associated with the application running on the virtual machine which interacts with the native code method for the selected call. Obtaining an intermediate representation associated with the application is described on page 8, lines 14-20 of the instant application.

The method further involves integrating the intermediate representation for the native code method for the selected call into the intermediate representation associated with the application running on the virtual machine to form an integrated intermediate representation. Integrating the intermediate representation for the native code method for the selected call into the intermediate representation associated with the application is described on page 8, lines 21-26 of the instant application.

The method further involves generating a native code from the integrated intermediate representation. Generating the native code from the integrated intermediate representation involves optimizing interactions between the application running on the virtual machine and the native code method for the selected call. Optimizing these interactions involves optimizing calls from the application to the native code method for the selected call by using additional information from the integrated intermediate representation to reduce a number of indirect calls and indirect references associated with the calls from the application to the native code method for the selected call. Generating the native code from the integrated intermediate representation is described on page 9, lines 1-4 of the instant application. The optimization process is described on page 7, lines 5-7, and page 9, lines 5-12 of the instant application.

Dependent Claim 2: A Method For Reducing An Overhead Involved In Executing Native Code Methods In An Application Running On A Virtual Machine

Dependent claim 2 depends upon independent claim 1. In the claimed method, selecting the call to any native code method involves selecting the call based upon at least one of: an execution frequency of the call, and an overhead involved in performing the call as compared against an amount of work performed by the native code method for the call. This selection process is described on page 7, lines 19-27 of the instant application.

Dependent Claim 4: A Method For Reducing An Overhead Involved In Executing Native Code Methods In An Application Running On A Virtual Machine

Dependent claim 4 depends upon independent claim 1. In the claimed method, optimizing interactions between the application running on the virtual machine and the native code method for the selected call involves optimizing callbacks by the native code method for the selected call into the virtual machine. Optimizing interactions in this manner is described on page 7, lines 5-7 of the instant application.

<u>Dependent Claim 5: A Method For Reducing An Overhead Involved In Executing Native Code Methods In An Application Running On A Virtual Machine</u>

Dependent claim 5 depends upon dependent claim 4. In the claimed method, optimizing callbacks by the native code method for the selected call into the virtual machine involves optimizing callbacks that access heap objects within the virtual machine. An object heap is described on page 6, lines 16-18 of the instant application.

<u>Dependent Claim 6: A Method For Reducing An Overhead Involved In Executing Native Code Methods In An Application Running On A Virtual Machine</u>

Dependent claim 6 depends upon dependent claim 4. In the claimed method, the virtual machine is a platform-independent virtual machine. Also, integrating the intermediate representation for the native code method for the selected call with the intermediate representation associated with the application running on the virtual machine involves integrating calls provided by an interface for accessing native code into the native code method for the selected call.

A platform-independent virtual machine is described on page 6, lines 3-15 of the instant application. An interface for accessing native code into the native code method for the selected call is described on page 6, line 19, to page 7, lines 4 of the instant application.

<u>Dependent Claim 7: A Method For Reducing An Overhead Involved In Executing Native Code Methods In An Application Running On A Virtual Machine</u>

Dependent claim 7 depends upon independent claim 1. In the claimed method, obtaining the intermediate representation associated with the application running on the virtual machine involves recompiling a corresponding portion of the application. Recompiling a portion of the application is described on page 8, lines 16-20 of the instant application.

Dependent Claim 8: A Method For Reducing An Overhead Involved In Executing Native Code Methods In An Application Running On A Virtual Machine

Dependent claim 8 depends upon independent claim 1. In the claimed method, obtaining the intermediate representation associated the application running on the virtual machine involves accessing a previously generated intermediate representation associated with the application running on the virtual machine. Accessing the previously generated intermediate representation is described on page 8, lines 16-20 of the instant application.

Dependent Claim 9: A Method For Reducing An Overhead Involved In Executing Native Code Methods In An Application Running On A Virtual Machine

Dependent claim 9 depends upon independent claim 1. In the claimed method, prior to decompiling the native code method for the selected call, the method further comprises setting up a context for the decompilation by determining a signature of the selected call, and determining a mapping from arguments of the selected call to corresponding locations in a native application binary interface (ABI). Setting up the context for the decompilation is described on page 8, lines 1-4 of the instant application.

<u>Independent Claim 10 and Dependent Claims 11, and 13-18: A Computer-Readable Storage Device Storing Instructions That When Executed By A Computer Cause The Computer To Perform A Method For Reducing An Overhead Involved In Executing Native Code Methods In An Application Running On A Virtual Machine</u>

Much of the subject matter taught in independent claim 1 and dependent claims 2, 4-9 also appears in independent claim 10 and dependent claims 11, and 13-18, respectively, as applied to a computer-readable storage device. Aside from the computer-readable storage device, which is described on page 5, lines 11-19, the remaining subject matter of claims 1-2, and 4-9, as summarized above, is sufficient to establish patentability. Appellant therefore does not repeat the above description.

<u>Independent Claim 28: A Method For Reducing An Overhead Involved In Executing Native Code Methods In An Application Running On A Virtual Machine</u>

The claimed method ("the second method") involves deciding to optimize a callback to any native code method into the virtual machine. Deciding to optimize a callback is described on page 7, lines 12-27 of the instant application.

Much of the subject matter taught in independent claim 1 also appears in the remainder of independent claim 28. The subject matter of claim 1, as summarized above, is sufficient to establish patentability. Appellant therefore does not repeat the above description.

Dependent Claim 29: A Method For Reducing An Overhead Involved In Executing Native Code Methods In An Application Running On A Virtual Machine

The second method further involves generating the native code from the integrated intermediate representation by optimizing calls by the application to the native code method for the callback. Optimizing calls by the application to the native code method is described on page 8, line 23, to page 9, line 4 of the instant application.

Dependent Claims 30-31: A Method For Reducing An Overhead Involved In Executing Native Code Methods In An Application Running On A Virtual Machine

Much of the subject matter of dependent claim 5-6 also appears in dependent claims 30-31. The subject matter of claims 5-6, as summarized above, is sufficient to establish patentability. Appellant therefore does not repeat the above description.

Independent Claim 32 and Dependent Claims 33-35: A Computer-Readable Storage Device Storing Instructions That When Executed By A Computer Cause The Computer To Perform A Method For Reducing An Overhead Involved In Executing Native Code Methods In An Application Running On A Virtual Machine

Much of the subject matter taught in independent claim 28 and dependent claims 29-31 also appears in independent claim 32 and dependent claims 33-35, respectively, as applied to a computer-readable storage device. Aside from the computer-readable storage device, which is described on page 5, lines 11-19, the remaining subject matter of claims 1-2, and 4-9, as summarized above, is sufficient to establish patentability. Appellant therefore does not repeat the above description.

GROUNDS OF REJECTION PRESENTED FOR REVIEW

In the Official Action mailed on 12 November 2009 (hereinafter "1109 OA"), Examiner reviewed claims 1-2, 4-11, 13-18, and 28-35. Examiner rejected claims 1-2, 4-7, 10-11, 13-16 and 28-35 under 35 U.S.C. § 103(a) as being unpatentable over Kwong et al. (U.S. patent no. 6,289,506, hereinafter "Kwong") in view of Ghosh (U.S. patent no. 6,412,109). Examiner rejected claims 8 and 17 under 35 U.S.C. § 103(a) as being unpatentable over Kwong and Ghosh in view of Kilis (U.S. patent no. 5,491,821). Examiner rejected claims 9, and 18 under 35 U.S.C. § 103(a) as being unpatentable over Kwong and Ghosh in view of Evans et al. (U.S. patent no. 5,805,899, hereinafter "Evans").

For the purposes of this appeal, and without admission as to the appropriateness of the other grounds raised by Examiner, Appellants will address Examiner's reliance on Kwong in rejecting independent claims 1, 10, 28, and 32. More specifically, Appellant will address Examiner's reliance on Kwong for the disclosure of de-compiling any native code method into an intermediate representation. Also, Appellant will address the appropriateness of the proposed modification of the Kwong prior art.

ARGUMENTS

Rejections under 35 U.S.C. § 103(a)

In response to the rejection under 35 U.S.C. § 103(a) in the 1109 OA, Appellant respectfully submits that the rejections cannot be sustained because:

(1) When establishing a prima facie case when rejecting claims under 35 U.S.C. § 103, Examiner's cited prior art must cover the claimed subject matter. Where the prior art does not cover the claimed subject matter, Examiner is required to explain the differences:

> The prior art reference (or references when combined) need not teach or suggest all the claim limitations, however, **Office personnel must explain why the difference(s) between the prior art and the claimed invention would have been obvious to one of ordinary skill in the art**; and
>
> The gap between the prior art and the claimed invention **may not be so great as to render the claim nonobvious to one reasonably skilled in the art.**[1]

In the instant case, the gap between the prior art cited by Examiner and the claimed invention is so great as to render the claims nonobvious to one reasonably skilled in the art; and

(2) When rejecting claims under 35 U.S.C. § 103, Examiner must not change the principle of operation of a reference:

> If the proposed modification or combination of the prior art would **change the principle of operation of the prior art invention being modified**, then the teachings of the references are not sufficient to render the claims prima facie obvious.[2]

[1] see MPEP § 2141(III), emphasis added
[2] see MPEP § 2143.01(VI), emphasis added

In the instant case, the modification of the prior art proposed by Examiner would change the principle of operation of the prior art invention being modified.

Overview of the Kwong System

In the interest of clarifying the following arguments, Appellant first provides an overview of the Kwong system.

Kwong discloses a system for optimizing Java performance using precompiled code.[3] As described by Kwong (and generally known in the art), a Java virtual machine (JV) can interpret Java bytecodes for execution on a processor.[4] Also, bytecodes can first be translated (e.g., compiled) into native machine code, and then the native machine code for the bytecodes can be executed on the processor.[5] Native machine code can execute faster that interpreted bytecodes.[6] Hence, compiling bytecodes into native machine code can improve the performance of a Java program.

In describing the system, Kwong discloses monitoring Java program execution and selecting a list of program methods to be precompiled into native machine code.[7] The native machine code for these program methods can be stored into a dynamic linked library (DLL) that is subsequently used by the Java program to improve performance when these program methods are executed.[8]

More specifically, Kwong discloses a programmer analyzing a program's performance and selecting a set of Java program methods to be optimized.[9] This set of program methods is compiled into native machine code, and the native

---

[3] see Kwong, column 2, lines 37-38
[4] see *id.*, column 3, line 66, to column 2, line 9
[5] see *id.*, column 2, lines 10-19
[6] see *id.*, column 6, line 64, to column 7, line 9
[7] see *id.*, column 3, line 66, to column 4, line 8
[8] see *id.*
[9] see *id.*, column 8, lines 23-35

machine code is included in a DLL.[10] In the Kwong system, the program performance is then analyzed using the DLL with the native machine code program methods.[11] If the performance is not satisfactory, the programmer can compile additional program methods into native machine code, and include this code in the DLL. Also, the programmer can de-compile some of the native machine code from the DLL back to Java bytecode.[12] Kwong discloses that the programmer can revert from the native machine code to the bytecode if the native machine code does not present the desired performance:

> At step 735, the selected Java program methods are optimized and compiled into native processor code by a native Java compiler. Or alternatively, a user may decide to de-compile earlier native compiled code back to bytecode format. The de-compile process may be used for instance when a user determines that the native compiled code does not present the desired performance and the user wants to revert the native compiled code back to Java bytecode. A dynamic linked library (DLL) is created for the compiled native program methods at step 740.[13]

Kwong also discloses that the optimization process can be repeated until a desired program performance is obtained:

> However, a programmer may repeat these steps to further refine and optimize the program. The process of monitoring and compiling bytecode/de-compiling native code may be repeated until the desired performance is obtained.[14]

In other words, Kwong at most discloses compiling program methods in a Java program from bytecode to machine code and reverting from the machine code for the program methods back to bytecode.

[10] see *id.*, column 8, lines 35-38
[11] see *id.*, column 8, lines 38-50
[12] see *id.*
[13] see *id.*
[14] see *id.*

<u>Rejection of Independent Claims 1, 12, and 23</u>

Examiner rejected independent claims 1, 10, 28, and 32 under 35 U.S.C. § 103(a) as being unpatentable over Kwong in view of Ghosh. Appellant respectfully disagrees with the rejection. The rejection of independent claims 1, 10, 28, and 32 is improper because:

1. The gap between the Kwong prior art and the claimed invention is so great as to render the claims nonobvious to one reasonably skilled in the art; and
2. The proposed modification or combination of the Kwong prior art would change the principle of operation of the Kwong invention being modified.

Appellant addresses these points in the following sections.

<u>1. The Gap between the Kwong Prior Art and the Claimed Invention is so Great as to Render the Claims Non-Obvious to One Reasonably Skilled in the Art</u>

Examiner has failed to establish prima facie obviousness because Examiner has failed to explain fundamental differences between the cited Kwong art and independent claims 1, 10, 28, and 32 in the instant application. Specifically, Examiner has failed to explain how Kwong's disclosure of de-compiling native code from **program methods** into bytecode renders obvious the present invention's de-compiling **any** native code method into an **intermediate representation**.

In the 1109 OA, Examiner argues that Kwong discloses decompiling at least part of any native code method into an intermediate representation. As discussed earlier, Kwong discloses de-compiling **earlier compiled** native code back to bytecode format.[15] Kwong clearly discloses that program methods of the

[15] see Kwong, col. 8, line 38-40

Java program can be compiled to native code, and that this native code (for the Java program) can be de-compiled back to bytecode format:

> A programmer would first write a computer program in the **Java** programming language in step 705.[16]
>
> A list of **Java program methods** that may improve program performance if optimized is generated at step 720. If the programmer finds that the performance is satisfactory at step 725, then development of the program is done. However, if the programmer decides to try to improve performance, then at step 730, he may select some of the **Java program methods** on the candidate list from step 720 for optimization. At step 735, the **selected Java program methods** are optimized and compiled into native processor code by a native Java compiler. Or alternatively, a user may decide to de-compile earlier native compiled code **back** to bytecode format. The de-compile process may be used for instance when a user determines that the native compiled code does not present the desired performance and the user wants to revert the native compiled code **back** to Java bytecode.[17]

In other words, Kwong discloses selecting a program method (i.e., a method within the Java program), and compiling this method to a native processor code. Also, Kwong discloses reverting from a previously compiled program method (in the native processor code) to bytecode for the method. Thus, Kwong is limited to converting native code for a program method into bytecode for the program method. Kwong nowhere discloses decompiling **any** native code method into an intermediate representation.

In contrast, the claimed embodiments select a call to **any** native code method. The claimed embodiments decompile at least part of the native code method for the selected call into an intermediate representation. As discussed in Applicant's remarks filed on 22 August 2008, **native code methods can exist**

[16] see *id.*, column 8, lines 23-25; emphasis added
[17] see *id.*, column 8, lines 28-43; emphasis added

outside an application.[18] For example, a computing device can provide native code methods to perform low-level system functions:

> Virtual machine 102 additionally provides a native interface 110, such as the Java Native Interface (JNI), which facilitates calls to methods in native code 112 from applications running on virtual machine 102. Note that computing device 100 provides a number of native code methods for performing low-level system functions, such as I/O operations.[19]

As shown in FIG. 2 of the instant application, the computing device includes native code 112 which is separate from the application 104. In other words, in the claimed embodiments, the Java program does not include the native code.

Appellant respectfully points out that the claimed embodiments can decompile **any** native code method, i.e., a code method which is not included in the Java application/program. The Kwong system is fundamentally distinct from the claimed embodiments, because Kwong only discloses de-compiling a method **for the Java program** back into bytecode. In other words, the Kwong system is limited to methods included in the Java program/application.

Furthermore, Appellant respectfully points out that, as discussed in Appellant's remarks filed on 22 July 2009, the term **intermediate representation** has a specific meaning in the field of software compilers.[20] More specifically, an intermediate representation includes a data structure which can be optimized during intermediate compiler operation. As described by Ghosh (and generally known in the art), bytecode is essentially a machine instruction set:

> The bytecodes executed by the JVM are essentially a machine instruction set, and as will be appreciated by those of ordinary skill in the art, are similar to the assembly language of a computing machine.[21]

[18] see Appellant's remarks filed 22 August 2008, pages 12-13
[19] see instant application, page 6, lines 19-23; emphasis added
[20] see Appellant's remarks filed 22 July 2009, pages 13-14

[21] see Ghosh, col. 1, lines 36-39

Kwong at most discloses de-compiling machine code into bytecode, i.e., translating from one instruction set to another. Kwong nowhere discloses de-compiling native code methods into an intermediate representation, as this term is generally understood in the art.

Appellant respectfully points out that the rejection of claims 1, 10, 28, and 32 under 35 U.S.C. § 103(a) using Kwong is improper because Examiner has failed to explain the above-described differences between the cited Kwong prior art and independent claims 1, 10, 28, and 32 in the instant application.

2. The Proposed Modification or Combination of the Kwong Prior Art Would Change the Principle of Operation of the Prior Art Invention Being Modified

Appellant respectfully notes that Examiner has failed to establish prima facie obviousness because Examiner has attributed principles of operation to portions of the Kwong prior art that are nowhere disclosed in Kwong and that would change the principle of operation of Kwong. Specifically, Examiner's proposed combination of the Kwong and Ghosh systems would change the principle of operation of Kwong.

As discussed above, Kwong is expressly limited to translating from one instruction set (native code) to another (bytecode). On the other hand, Ghosh discloses using an intermediate representation that includes bytecode, as well as additional information:

> The IR provides information about two essential components of the program: the control flow graph (CFG) and the data flow graph (DFG).[22]
>
> **The CFG breaks the code into blocks of bytecode**, termed basic blocks, that are always performed as an uninterrupted group of instructions, and

[22] see Ghosh, col. 2, lines 17-20

establishes the connections that link the basic blocks together. In so doing, the CFG represents **different variations** of the sequence in which the instructions of a program can be performed. The connections between basic blocks are known in the art as edges. **The DFG maps the connections between where data values are produced and where they are used.**[23]

In other words, the Ghosh intermediate representation is fundamentally distinct from the bytecode of Kwong. As discussed in Appellant's remarks filed on 22 July 2009, the intermediate representation can be a data structure which can be optimized during an intermediate compiler operation, which is performed prior to generating the executable binary instructions from the intermediate representation.[24] This data structure can include additional information to reduce the number of indirect calls and indirect references associated with the calls.

Appellant respectfully points out that neither Kwong, nor Ghosh disclose de-compiling an intermediate representation into bytecode. Kwong merely discloses translating from one instruction set to another. Kwong nowhere discloses reverting from a representation that includes code, and additional information, back to bytecode. More specifically, the Ghosh intermediate representation can include **different variations** of the sequence in which the instructions of a program can be performed.[25] Kwong nowhere discloses de-compiling into bytecode a representation that includes these different variations of the sequence in which the instructions of a program can be performed.

Because Kwong nowhere discloses that the bytecode includes additional information, Examiner's proposed modification or combination of the prior art would change the principle of operation of the prior art invention of Kwong. Hence, the rejection of claims 1, 10, 28 and 32 under 35 U.S.C. § 103(a) based on Kwong in view of Ghosh is improper because Examiner has proposed a

[23] see *id.*, col. 2, lines 24-32; emphasis added
[24] see Applicant's remarks filed on 22 July 2009, pages 13-14
[25] see *id.*, col. 2, lines 24-32; emphasis added

modification of the Kwong prior art that was nowhere disclosed in Kwong, and that would modify the principle of operation of Kwong.

Conclusion

In summary, Appellant has demonstrated that the rejections of claims 1, 10, 28, and 32 under 35 U.S.C. § 103(a) based on Kwong in view of Ghoush are improper because Examiner has failed to explain fundamental differences between Kwong and the claimed embodiments, and because the differences amount to a gap that is sufficient to render the claimed embodiments non-obvious to one having ordinary skill in the art. In addition, Appellant demonstrated that the rejection is improper because the proposed modification of the Kwong prior art would change a principle of operation of Kwong.

In view of the foregoing, Appellant respectfully requests the reversal of the rejections of claims 1, 10, 28, and 32 in the 1109 OA. Appellant further requests allowance of claims 1-2, 4-11, 13-18, and 28-35.

Respectfully submitted,

By: /Anthony Jones/
Anthony Jones
Registration No. 59,521

Date: 19 May 2010

Anthony Jones
Park, Vaughan & Fleming LLP
2820 Fifth Street
Davis, CA 95618-7759
Tel: (530) 759-1666
Fax: (530) 759-1665
Email: tony@parklegal.com

APPENDICES

Appendix A: Claims Appendix

1. (Previously Presented) A method for reducing an overhead involved in executing native code methods in an application running on a virtual machine, comprising:

selecting a call to any native code method to be optimized within the virtual machine;

decompiling at least part of the native code method for the selected call into an intermediate representation, wherein an intermediate representation includes a set of instruction code which is not in final executable form;

obtaining an intermediate representation associated with the application running on the virtual machine which interacts with the native code method for the selected call;

integrating the intermediate representation for the native code method for the selected call into the intermediate representation associated with the application running on the virtual machine to form an integrated intermediate representation; and

generating a native code from the integrated intermediate representation, wherein generating the native code from the integrated intermediate representation involves optimizing interactions between the application running on the virtual machine and the native code method for the selected call, wherein optimizing the interactions involves optimizing calls from the application to the native code method for the selected call by using additional information from the integrated intermediate representation to reduce a number of indirect calls and indirect references associated with the calls from the application to the native code method for the selected call.

2. (Previously Presented) The method of claim 1, wherein selecting the call to any native code method involves selecting the call based upon at least one of:

an execution frequency of the call; and

an overhead involved in performing the call as compared against an amount of work performed by the native code method for the call.

3 (Canceled).

4. (Previously Presented) The method of claim 1, wherein optimizing interactions between the application running on the virtual machine and the native code method for the selected call involves optimizing callbacks by the native code method for the selected call into the virtual machine.

5. (Previously Presented) The method of claim 4, wherein optimizing callbacks by the native code method for the selected call into the virtual machine involves optimizing callbacks that access heap objects within the virtual machine.

6. (Previously Presented) The method of claim 4, wherein the virtual machine is a platform-independent virtual machine; and

wherein integrating the intermediate representation for the native code method for the selected call with the intermediate representation associated with the application running on the virtual machine involves integrating calls provided by an interface for accessing native code into the native code method for the selected call.

7. (Original) The method of claim 1, wherein obtaining the intermediate representation associated with the application running on the virtual machine involves recompiling a corresponding portion of the application.

8. (Original) The method of claim 1, wherein obtaining the intermediate representation associated the application running on the virtual machine involves accessing a previously generated intermediate representation associated with the application running on the virtual machine.

9. (Previously Presented) The method of claim 1, wherein prior to decompiling the native code method for the selected call, the method further comprises setting up a context for the decompilation by:

determining a signature of the selected call; and

determining a mapping from arguments of the selected call to corresponding locations in a native application binary interface (ABI).

10. (Previously Presented) A computer-readable storage device storing instructions that when executed by a computer cause the computer to perform a method for reducing an overhead involved in executing native code methods in an application running on a virtual machine, the method comprising:

selecting a call to any native code method to be optimized within the virtual machine;

decompiling at least part of the native code method for the selected call into an intermediate representation, wherein an intermediate representation includes a set of instruction code which is not in final executable form;

obtaining an intermediate representation associated with the application running on the virtual machine which interacts with the native code method for the selected call;

integrating the intermediate representation for the native code method for the selected call into the intermediate representation associated with the application running on the virtual machine to form an integrated intermediate representation; and generating a native code from the integrated intermediate representation, wherein generating the native code from the integrated intermediate representation involves optimizing interactions between the application running on the virtual machine and the native code method for the selected call, wherein optimizing the interactions involves optimizing calls from the application to the native code method for the selected call by using additional information from the integrated intermediate representation to reduce a number of indirect calls and indirect references associated with the calls from the application to the native code method for the selected call.

11. (Previously Presented) The computer-readable storage device of claim 10, wherein selecting the call to any native code method involves selecting the call based upon at least one of:

an execution frequency of the call; and

an overhead involved in performing the call as compared against an amount of work performed by the native code method for the call.

12 (Canceled).

13. (Previously Presented) The computer-readable storage device of claim 10, wherein optimizing interactions between the application running on the virtual machine and the native code method for the selected call involves optimizing callbacks by the native code method for the selected call into the virtual machine.

14. (Previously Presented) The computer-readable storage device of claim 13, wherein optimizing callbacks by the native code method for the selected call into the virtual machine involves optimizing callbacks that access heap objects within the virtual machine.

15. (Previously Presented) The computer-readable storage device of claim 13,

wherein the virtual machine is a platform-independent virtual machine; and

wherein integrating the intermediate representation for the native code method for the selected call with the intermediate representation associated with the application running on the virtual machine involves integrating calls provided by an interface for accessing native code into the native code method for the selected call.

16. (Previously presented) The computer-readable storage device of claim 10, wherein obtaining the intermediate representation associated with the application running on the virtual machine involves recompiling a corresponding portion of the application.

17. (Previously presented) The computer-readable storage device of claim 10, wherein obtaining the intermediate representation associated with the application running on the virtual machine involves accessing a previously generated intermediate representation associated with the application running on the virtual machine.

18. (Previously Presented) The computer-readable storage device of claim 10, wherein prior to decompiling the native code method for the selected call, the method further comprises setting up a context for the decompilation by:

determining a signature of the selected call; anddetermining a mapping from arguments of the selected call to corresponding locations in a native application binary interface (ABI).

19-27. (Cancelled)

28. (Previously Presented) A method for reducing an overhead involved in executing native code methods in an application running on a virtual machine, comprising:

deciding to optimize a callback by any native code method into the virtual machine;

decompiling at least part of the native code method for the callback into an intermediate representation, wherein an intermediate representation includes a set of instruction code which is not in final executable form;

obtaining an intermediate representation associated with the application running on the virtual machine which interacts with the native code method for the callback;

integrating the intermediate representation for the native code method for the callback into the intermediate representation associated with the application running on the virtual machine to form an integrated intermediate representation; and

generating a native code from the integrated intermediate representation, wherein generating the native code from the integrated intermediate representation involves optimizing the callback, wherein optimizing the callback involves optimizing calls from the native code method for the callback to the application by using additional information from the integrated intermediate representation to reduce a number of indirect calls and indirect references associated with the calls from the native code method for the callback to the application.

29. (Previously Presented) The method of claim 28, wherein generating the native code from the integrated intermediate representation also

involves optimizing calls by the application to the native code method for the callback.

30. (Previously Presented) The method of claim 28, wherein optimizing the callback by any native code method into the virtual machine involves optimizing a callback that accesses a heap object within the virtual machine.

31. (Previously Presented) The method of claim 28,

wherein the virtual machine is a platform-independent virtual machine; and

wherein integrating the intermediate representation for the native code method for the callback with the intermediate representation associated with the application running on the virtual machine involves integrating calls provided by an interface for accessing native code into the native code method for the callback.

32. (Previously Presented) A computer-readable storage device storing instructions that when executed by a computer cause the computer to perform a method for reducing an overhead involved in executing native code methods in an application running on a virtual machine, the method comprising:

deciding to optimize a callback by any native code method into the virtual machine;

decompiling at least part of the native code method for the callback into an intermediate representation, wherein an intermediate representation includes a set of instruction code which is not in final executable form;

obtaining an intermediate representation associated with the application running on the virtual machine which interacts with the native code method for the callback;

integrating the intermediate representation for the native code method for the callback into the intermediate representation associated with the application running on the virtual machine to form an integrated intermediate representation; and

generating a native code from the integrated intermediate representation, wherein generating the native code from the integrated intermediate representation involves optimizing the callback, wherein optimizing the callback involves optimizing calls from the native code method for the callback to the application by using additional information from the integrated intermediate representation to reduce a number of indirect calls and indirect references associated with the calls from the native code method for the callback to the application.

33. (Previously Presented) The computer-readable storage device of claim 32, wherein generating the native code from the integrated intermediate representation also involves optimizing calls by the application to the native code method for the callback.

34. (Previously Presented) The computer-readable storage device of claim 32, wherein optimizing the callback by any native code method into the virtual machine involves optimizing a callback that accesses a heap object within the virtual machine.

35. (Previously Presented) The computer-readable storage device of claim 32, wherein the virtual machine is a platform-independent virtual machine; and

wherein integrating the intermediate representation for the native code method for the callback with the intermediate representation associated with the application running on the virtual machine involves integrating calls provided by an interface for accessing native code into the native code method for the callback.

36-39. (Canceled)

Appendix B: Evidence

For this appeal, Appellants do not rely on any evidence submitted pursuant to §§ 1.130, 1.131, or 1.132, or other evidence entered by Examiner.

Appendix C: Related Proceedings

Appellants are aware of no related proceedings.